Hierarchical Multi-Layer Screen-Space Ray Tracing

Nikolai Hofmann Phillip Bogendörfer Marc Stamminger Kai Selgrad
Computer Graphics Group, University of Erlangen-Nuremberg

Figure 1: With our hierarchical multi-layer ray tracer, advanced effects, especially from long rays, can be rendered efficiently. Both images shown demonstrate two-bounce screen-space reflections. Note the reflection of the red and blue curtains in the left-most part in Sponza and the reflections in the windows of San Miguel. Render times for multi-layer framebuffer generation are 1.34 ms, computing our hierarchy takes 1.98 ms and tracing two reflections bounces 4.93 ms (left), and render 5.62 ms, hierarchy 2.09 ms and tracing two bounces 5.25 ms (right). All at 720p with 4 layers on Geforce 1070 GTX without any further common optimizations such as sub-sampling or increased strides.

ABSTRACT

In this paper we present a method for fast screen-space ray tracing. Single-layer screen-space ray marching is an established tool in high-performance applications, such as games, where plausible and appealing results are more important than strictly correct ones. However, even in such tightly controlled environments, missing scene information can cause visible artifacts. This can be tackled by keeping multiple layers of screen-space information, but might not be afforable on severely limited time-budgets. Traversal speed of single-layer ray marching is commonly improved by multi-resolution schemes, from sub-sampling to stepping through mip-maps to achieve faster frame rates. We show that by combining these approaches, keeping multiple layers and tracing on multiple resolutions, images of higher quality can be computed rapidly. Figure 1 shows this for two scenes with multi-bounce reflections that would show strong artifacts when using only a single layer.

CCS CONCEPTS

•Computing methodologies →Ray tracing; Rasterization;

KEYWORDS

Screen-Space Ray Tracing, Multi-Layer, Real-Time Rendering

HPG '17, Los Angeles, CA, USA

978-1-4503-5101-0/17/07…$15.00
DOI: 10.1145/3105762.3105781

ACM Reference format:
Nikolai Hofmann, Phillip Bogendörfer, Marc Stamminger, and Kai Selgrad. 2017. Hierarchical Multi-Layer Screen-Space Ray Tracing. In *Proceedings of HPG '17, Los Angeles, CA, USA, July 28-30, 2017,* 10 pages.
DOI: 10.1145/3105762.3105781

1 INTRODUCTION

Ray tracing in screen-space is a common approach to provide advanced effects in real-time contexts. In contrast to traditional, geometry-based ray traversal, the approximation of tracing rays against screen-space fragments is appealing when the time budget allocated for these effects is severely limited. One prominent example is computing screen-space reflections, a technique that has become one of the standard assets in real-time applications, such as games [Sikachev 2014; Stachowiak and Uludag 2015; Valient 2014; Wronski 2014], since its introduction [Graham 2010; Sousa et al. 2011]. One of the fundamental issues with screen-space ray tracing is that objects not visible from the camera cannot influence the image, which can lead to noticeable image artifacts. In tightly controlled settings this can be managed manually by careful placement of objects and selection of materials, but the general case requires more data than just the front-most visible fragments. By providing more information to the algorithm, artifacts can be reduced, but usually at comparatively high cost during rendering.

Common optimizations for screen-space ray marching include sub-resolution schemes (such as stepping in mip-maps or skipping pixels along the ray), as well as (in most cases) ignoring multiple layers. To this end we present a simple screen-space acceleration structure that tracks multiple layers of the scene and greatly speeds up ray traversal, especially for long rays that occur, e.g., with

reflections. That is, out method combines having higher-quality multi-layer information with a sub-resolution representation in a unified way. As input we only consider the fragments rasterized for the camera's view. Therefore, our approach does not target support for completely omnidirectional reflections, it only operates in the view-frustum. This is a common limitation [Mara et al. 2014, 2016; McGuire and Mara 2014; Sikachev 2014; Sousa et al. 2011; Stachowiak and Uludag 2015; Valient 2014; Wronski 2014] that could be lifted by allowing more time.

Our acceleration structure can be built during rendering and provides very fast ray queries when used with our hierarchical screen-space traversal. Instead of only keeping the front-most fragments [Sousa et al. 2011; Stachowiak and Uludag 2015; Valient 2014; Wronski 2014] we construct a per-pixel linked list [Yang et al. 2010]. Note that our scheme could also be used with efficient depth-peeling approaches, such as Mara et al.'s [2016]. From the thusly collected fragments we can then build a spatial 2.5D hierarchy of bounding volumes (in form of non-overlapping intervals). Ray traversal in screen-space is then executed in a DDA [Amanatides and Woo 1987; Fujimoto et al. 1986] fashion, i.e. by conservative, 2D line rasterization, but on multiple resolutions, and traversing multiple intervals at each visited screen-space location.

Contribution. In this paper we present an extension of hierarchical ray marching [Stachowiak and Uludag 2015] that uses multiple layers of scene information [Mara et al. 2014]. We introduce an efficient, hierarchical DDA-style ray marching algorithm that relies on a dynamic data structure generated during scene rendering. Our ray traversal is faster than previous DDA-based approaches [Mara et al. 2016; McGuire and Mara 2014] and amenable to common optimizations applied in real-world settings [Sousa et al. 2011; Stachowiak and Uludag 2015; Valient 2014; Wronski 2014] (having one of them, hierarchical ray marching, as its corner stone). Specifically, the contributions presented in this paper are:

- A simple multi-resolution scheme to efficiently build a dynamic screen-space acceleration structure from multi-layered buffers.
- An efficient, multi-resolution extension of the DDA ray marching scheme that traverses multiple layers of geometry for each visited pixel and allows to skip large areas of empty space.
- A description of how to efficiently trace multiple bounces (or samples) and demonstration of that at real-time frame rates.

In our implementation we use per-pixel linked list [Yang et al. 2010] to generate the multi-layered base level, but we note that different approaches, e.g. relying on deep GBuffers [Mara et al. 2016], are also valid starting points.

Figure 2 shows a simple scene rendered with our approach. The limits of screen-space reflections are clearly visible: missing information for the ceiling and sporadic ray misses in indirectly visible surfaces. Nevertheless, the figure shows three-bounce screen-space reflections in a challenging setup and still provides appealing results. See Figure 1 for more realistic applications.

In the remainder of this paper we will first give a brief overview of related work (Section 2), followed by an overview of our hierarchy and traversal method (Section 3). We then describe the individual steps of our algorithm, along with relevant details for fast GPU implementation (Sections 4 and 5). We further evaluate our method's performance and the quality of its results with respect to a recent, high-performance method (Section 6) and conclude by summarizing our findings and listing future work (Section 7).

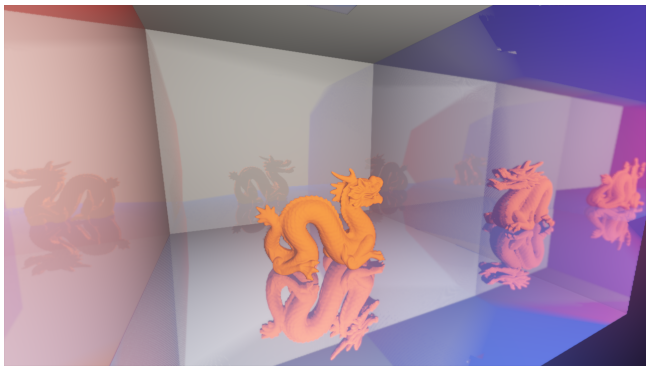

Figure 2: Dragon in a mirror-box with three-bounce screen-space reflections. A very challenging, albeit technical setup.

2 RELATED WORK

In this section we cover related work on multi-layer approaches to approximate global illumination effects, screen-space ray tracing, and real-time GPU ray tracing to provide context for our work. As this is a very broad spectrum we limit our treatment to the most closely related references.

Mulit-Layer Approximations. Storing multiple layers of geometry found to overlap a pixel during rasterization provides a richer data-set that can be used to avoid artifacts and provides more scene-information to methods working in image space [Shade et al. 1998]. Early implementations of global illumination approximations on programmable GPUs used only few, separately rendered layers. Examples include Wyman's work on refractions [2005a; 2005b] and Lee et al.'s handling of disocclusion in depth-of-field (DOF) rendering [2008]. More complete layering information can be obtained by depth peeling [Bavoil and Myers 2008; Everitt 2001] or concurrent list construction [Yang et al. 2010]. Apart from transparency [Bavoil and Myers 2008; Everitt 2001; Yang et al. 2010] such data structures have been successfully applied to compute ambient occlusion [Bavoil and Myers 2008; Mara et al. 2016; Ritschel et al. 2009] local indirect illumination [Mara et al. 2016; Ritschel et al. 2009], screen-space reflections and refractions [McGuire and Mara 2014] and DOF [Lee et al. 2009, 2010] with high quality. This quality, however, is usually traded for an increase of computation time.

Nichols and Wyman [2009] show how compute-intensive screen-space algorithms can benefit from multi-resolution approaches. Multi-layer filtering, a multi-resolution approach that filters multi-layer structures similar to mip-maps, has been shown to support fast rendering of DOF [Selgrad et al. 2015] and soft shadows [Selgrad et al. 2014].

A different approach to these problems was presented by Nalbach et al [2014], where the scene-geometry is tessellated on-the-fly to generate surfels [Pfister et al. 2000] which are then splatted to compute indirect illumination, ambient occlusion and subsurface scattering.

Screen-Space Ray Tracing. Early screen-space ray tracing by height-map bisection, such as with Lee et al.'s method to compute depth-of-field [2009], assume that the scene's depth structure can be partitioned into disjoint intervals. Such height-field representations are, however, usually only crude approximations, missing hidden surfaces inside a single layer. Therefore, they provide fast computation, but are prone to missing important scene elements. This is especially the case for reflective, refractive and ambient occlusion rays that are not as perpendicular to the height-field as DOF rays.

Recently, single-layer screen-space ray tracing by simple ray marching [Sousa et al. 2011] has become a standard effect seen in games [Sikachev 2014; Sousa et al. 2011; Stachowiak and Uludag 2015; Valient 2014; Wronski 2014]. McGuire and Mara [2014] showed how perspective correct ray marching can avoid common problems such as over- and undersampling. However, keeping track of only a single depth-layer is problematic as important scene information will be missed [Ganestam and Doggett 2015; Mara et al. 2014]. In production settings, common optimizations for screen-space ray marching are sub-sampling [Stachowiak and Uludag 2015; Valient 2014; Wronski 2014], stepping in mip-maps [Stachowiak and Uludag 2015; Valient 2014] and increasing the step-size [McGuire and Mara 2014; Stachowiak and Uludag 2015; Valient 2014].

Mara et al. [2014] proposed a temporal improvement to depth-peeling and keep only a few depth-layers which are put to best use by requiring a certain minimum-separation between those layers. They further proposed an improvement of this depth-peeling step by using geometry-shader extensions [2016]. Ganestam and Doggett [2015] instead follow a hybrid approach, partitioning the scene into a limited near-field area where fully dynamic geometry-based GPU ray tracing is used, and a single-layer per-frame cube map to approximate reflection and refraction rays in arbitrary directions by screen-space ray marching. Widmer et al. [2015] further propose to build a quad-tree on the fragments generated for each cube-map face. This quad-tree holds all the fragments rasterized (for cube maps), not only the front-most ones, and can further approximate almost-flat regions with leaf-nodes (similar to multi-layer filtering [Selgrad et al. 2014]).

Fast GPU Ray Tracing. With ever faster GPU ray traveral [Aila et al. 2012] and acceleration structure construction methods [Karras 2012; Lauterbach et al. 2009] the option to drop geometric approximations and rely on real ray tracing instead is becoming ever more realistic. This holds even more so in hybrid settings [Ganestam and Doggett 2015]. Findings from this context can also be applied to speed up approximations, as we show in Section 5.3.

3 ALGORITHM OVERVIEW

We start by giving a high-level overview of our algorithm to provide context for the more in-depth description of the individual steps in the following sections.

Our algorithm works by keeping the k front-most layers of fragments rasterized inside the view-frustum. This is more information than common screen-space ray tracers employ [Sousa et al. 2011; Stachowiak and Uludag 2015; Valient 2014; Wronski 2014], but not as comprehensive as with multi-view approaches [Widmer et al. 2015] or classical ray tracing.

We generate this set of fragments with a single pass over the scene geometry that collects per-pixel linked lists [Yang et al. 2010], followed by a sorting pass. Note that, as we will use this data-set to approximate light transport in different settings, we have to store enough information about the hit-point to compute shading and reflection values on.

Since the amount of fragments rasterized is unknown in advance, and may change per frame and scene, it is recommended to conservatively allocate memory in order to be capable of holding all collected fragments. Even though potentially wasteful, overallocating might be sensible as costly reallocation at run-time should be avoided and, since the thread execution order on GPUs is non-deterministic, front-most fragments may be collected last. It is also possible to detect that storage has run out, or to query the actual amount of memory consumed, by reading back the atomic counter used for linked-list creation. Note that such issues can be avoided with depth-peeling based methods [Mara et al. 2016] which are an equally valid base for our algorithm.

The sorting-pass then reduces the fully collected lists (up to the scene's depth complexity) to the k front-most fragments, optionally using minimum-separation [Mara et al. 2014]. To ensure stable results, the entire list of fragments for each pixel has to be traversed and (partially) sorted, even when only a fraction will be written back again.

When tracing through fragments of multiple layers each fragment needs to have an associated depth-extent [McGuire and Mara 2014]. This is important to determine if a ray hit a fragment in question, or travels behind it. We add this information to the sorted fragments and write back a z-interval for each input fragment kept.

Based on these intervals we generate lower-resolution representations by merging similar intervals from neighboring pixels and in depth-direction. This way we arrive at an hierarchy of intervals that, together with the sub-resolution pixel dimensions, specify a spatial hierarchy. This hierarchy could also be traced in a more traditional sense, similar to the approach by Widmer et al. [2015], however, our approach is geared towards DDA-style traversal.

To this end we employ a multi-resolution DDA method. Starting from a location found during rendering (or by previous bounces in our structure) we project a point along the ray's direction into screen-space and, after proper clipping, find the screen-space direction to use. With this starting point and direction we follow Amanatides and Woo's [1987] conservative traversal and walk the screen-space pixel raster by keeping track of the closest neighbor to traverse next. This is executed on different resolutions as follows: We remain on the finest level of the hierarchy for a few steps before progressively switching to lower-resolution levels with intervals spanning larger areas of space. As long as no intersections through evaluating ray and interval depth are found, traversal continues on coarse resolutions. When an interval overlapping the ray is found, then the traversal state is advanced to the location where the ray enters this interval and the DDA switches to a higher resolution before continuing. Hit-points are only found on the finest level, i.e. the intervals are not used for hit-point approximation (as by Selgrad et al. [2014]), but only as an acceleration structure.

In the following we describe these steps in more detail, together with information about how to efficiently implement this scheme on modern GPUs.

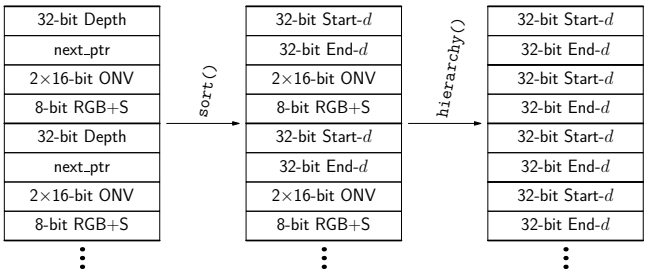

Figure 3: Data-format of our per-pixel collected lists/arrays: collected lists from rendering (left), sorted lists converted to interval-arrays (center) and compacted interval-hierarchy arrays (right).

4 HIERARCHY CONSTRUCTION

In this section we will describe how our hierarchy is built, starting from scene rendering to collect all fragments in the view-frustum (Section 4.1) over sorting (Section 4.2) to generating multi-resolution intervals (Section 4.3).

4.1 Base-Level Generation

With our method, we render the scene geometry only once. We can either choose to shade each fragment as it is rasterized, or only keep color information for deferred shading. The best choice depends on how often shading will have to be computed for a given ray tracing task. For example, with reflections shading will have to be computed for the front-most face as well as for each bounce traced. If this is significantly less that the number of fragments collected, then deferred shading will be more efficient. Unless otherwise noted this is our default choice.

We also need to store the surface normals for further processing (e.g. for reflections, refraction, AO), but want to keep bandwidth as low as possible. Therefore, we use octahedron normal vectors (ONV) [Meyer et al. 2010] to encode them in two 16-bit values. Furthermore, we have to store color (either shaded, or material) and depth-values. The former are stored with three 8-bit values for the diffuse color, a single channel specular value and 32-bit depth. Including the pointer for the list-structure, each node is 128-bit, that is, a single four-component vector (see Figure 3, left). Storing the data this compactly benefits performance as our algorithm is, due to heavy list traversal, bandwidth-bound. If higher-precision color values are required the options are to either compromise on the quality of the normal or depth-values, or to incur a small performance penalty for keeping an extra array. Should only a few bits be required to enhance the color-range, they could be shaved off the next-pointer without imposing any practical limit on fragment counts. Note that for physically based rendering, which is common in recent game engines, more material properties may be required. Since our algorithm is already bandwidth-bound and more parameters would further increase bandwidth, it should be attempted to stay within 16 Bytes per fragment by shaving bits off the depth and next pointer. Otherwise, however, a performance penalty is to be expected. This is a limitation of the linked list approach, where depth-peeling based approaches simply perform better due to requiring less bandwidth.

4.2 List Truncation and Conversion

One way to improve the memory-access characteristics of the following list merging and traversal steps is to convert from lists to sequentially stored per-pixel arrays. We do this by preallocating a larger list-buffer and placing the arrays at the back of the list-data. In terms of memory-consumption this is somewhat wasteful (see Section 6), however, regarding run-time this conversion provides a significant speed-up in the later stages of the algorithm. Even though it does not help with bandwidth, it avoids latency-bottlenecks and ensures more coherent access patterns in the subsequent stages that make heavy use of this data. Note that the conversion itself is free as we have to re-write the data after sorting and truncation to k layers, regardless of this choice. Also note that due to discarding geometry from overly long lists, artifacts may be introduced (see Section 6).

This improvement has been noted before [Selgrad et al. 2015], but only in conjunction with scanning-passes. In our method we accept to over-allocate and simply assume that each pixel requires all k layers. This way, indexing is coherent and, more importantly, without any synchronization-overhead. Section 6 puts this overhead in relation with the overall memory requirements of our method.

For sorting we use straightforward insertion sort (as also suggested by Yang et al. [2010]) where we keep the partially sorted arrays in per-thread shared memory to avoid register spills. We only need k cells as farther-away fragments are not kept and can be shifted out of the array while sorting. The number of valid entries stored in the thusly sorted per-pixel arrays is stored along the offset-pointer (head-pointer of the list-data).

We usually collect between 4 to 8 layers of fragments for each pixel on the base-level, discarding any left-over fragments farther behind. To ensure better depth-coverage of those samples we can optionally follow McGuire and Mara [2014] and ensure minimum-separation, i.e. that two successive fragments have at least a certain distance. However, while successfully thinning out dense areas, this can lead to missing important intersections when tracing. A more robust way is to adapt the fragment-thickness in such cases and merge multiple fragments during the sorting step. As we store intervals in the first place, variable fragment-thickness maps well to our approach. However, even though adaptive thickness reduces the likelihood of misses, it cannot easily ensure that the correct color (or normal) is returned as appropriate filtering is not easily possible in such cases. Figure 3 (center) illustrates the data-format after sorting, where the list-pointer is replaced with the z-thickness (either fixed, scene-dependent [McGuire and Mara 2014] or adaptive after merging).

4.3 Interval Resolution-Pyramid

The construction of our hierarchy is implemented in a process similar to computing mip-maps: we merge the intervals stored in neighboring 2×2 regions. This is a very simple, but bandwidth-intensive process. We load the first interval of each pixel from the previous level and find the front-most one. This interval is then extended with successive intervals from the 2×2 neighborhood (reloading new data as necessary) until its extent exceeds a given threshold τ. At this point we emit the interval to the current level and then start generating a new one. That is, to compute the first

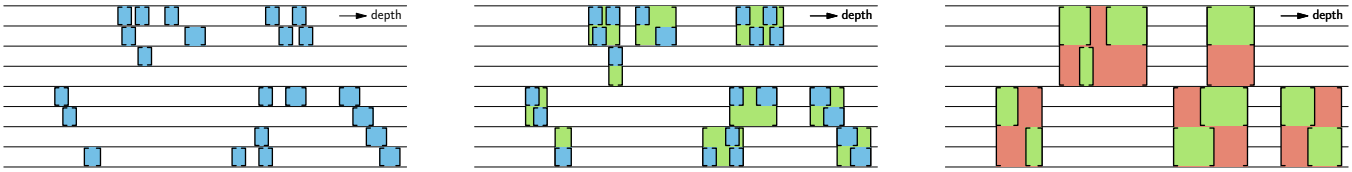

Figure 4: Three levels of merged interval-resolutions. The intervals shown in the left image are successively merged across fragment-neighborhoods and z-distance. The horizontal lines show the spatial extent of the per-pixel arrays of the base level.

interval $I^L_{x,y,0}$ for target-pixel x, y on level L, we start from depth $d_{\min} = 0$ until $d_{\max} = d_{\min} + \tau$ by merging all intervals of the 2×2 region (on level $L-1$) in-between $d_{\min}$ and $d_{\max}$:

$$I^L_{x,y,\text{new}} = \bigcup I^{L-1}_{x',y',j}$$

with $x' \in \{2x, 2x+1\}$ and $y' \in \{2y, 2y+1\}$, $\min(I^{L-1}_{x',y',j}) \geq d_{\min}$ and $\max(I^{L-1}_{x',y',j}) \leq d_{\max}$. This process is repeated as long as there are intervals left, generally resulting in a decrease of intervals as we further relax τ over successive levels. The rationale behind this is that we also merge over a larger pixel-footprint and can thus allow larger depth-intervals to be combined. We generally set τ to 25 times the projected pixel-footprint at the starting-depth of the interval in question, but note that this parameter is very robust over a wide range (15 to 35 for Figure 1, left).

Figure 4 illustrates this merging process over three levels. The left image shows the depth-intervals (in blue) for fragments generated for 8 pixels (aligned vertically). The center image shows the same data, but overlaid (in green) with the intervals generated by merging similar intervals from the base level. As can be seen, merging occurs over depth as well as pixel neighborhood. The right image shows the results of yet another merging pass (overlaid in red), based on the previous merge (shown in the center).

Note that τ only affects the quality of our acceleration-structure, not the quality of the results: we use the interval-hierarchy only to skip over empty space. Having too small a threshold will lead to long arrays that have to be traversed before finding out whether there is an intersection with a ray in a given list. Having too large a threshold will lead to very short lists, yielding many unnecessary intersections with a ray.

In the first invocation of this merging step we convert our data-representation as generated in the sorting stage to a more compact version (see Figure 3, right). As the interval-hierarchy is a true acceleration-structure, the normal and color information is not required when leaving the base level. To this end we collapse the four-component vectors required on the base level to two-component vectors only holding the intervals' entry and exit depths.

5 INTERVAL-HIERARCHY RAY-MARCHING

For screen-space ray marching it is a common choice to follow a Bresenham-style traversal scheme [McGuire and Mara 2014; Wronski 2014]. In single-level settings this is a reasonable choice as missed pixels only have a small impact on image quality. Our approach, on the other hand, would suffer much from such a traversal method as missed pixels on a coarser level of the structure can map to large screen-space areas. Therefore, we rely on the more exact, but arithmetically more complex, DDA-scheme proposed by Amanatides and Woo [1987]. Section 6 shows that the hit-points computed are a little more exact, but the primary motivation is to avoid more severe misses on higher levels of our hierarchy.

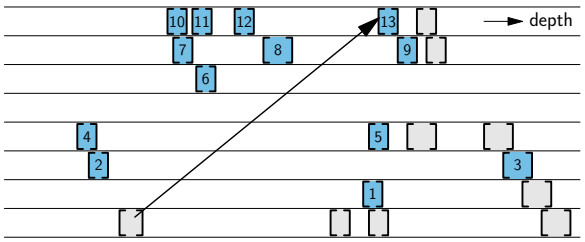

Figure 5: Single-level traversal of fragments. The gray intervals are not considered during traversal as an earlier one has been found to be behind the ray. Traversal starts at the head of the list for each pixel visited by the DDA.

5.1 Single-Level Traversal

We start our traversal by determining screen-space start and end-points for the ray. To this end we re-project the starting position in screen-space and compute the (application specific) ray direction in eye-space. We then clip and project a suitable ending position along the ray back to screen-space. We keep homogeneous interpolation values according to McGuire and Mara [2014] to ensure perspective correct interpolation of the ray's eye-space position.

With this we intersect the current pixel's bounding box with the screen-space ray to determine the exit-parameters (in x and y) along the ray. We choose the lower of these and update our traversal state to the newly selected pixel. This skips the first pixel to avoid getting stuck in self-intersections.

For further pixels along the ray we also intersect their bounding box with the screen-space ray to determine ray entry- and exit parameters in x and y, i.e. t_0 and t_1. This corresponds to a simple, 2D ray/AABB test, including possible exchange of the min/max values depending on the ray direction. Using perspective correct interpolation we can map the pixel entry- and exit parameters to depth values as stored in our interval-arrays, $z_i = \text{interpolate}(t_i)$. With these z-bounds we traverse the intervals stored for the current pixel until we find one overlapping $[z_0, z_1]$, which will then be returned as the intersection result. Note that, as intervals do not overlap (neither in x, y, nor in z) this yields a single fragment (see Section 4.1). As soon as an interval starts farther than z_1, traversal continues according to the earlier pixel-bounding box intersection. Figure 5 illustrates this traversal scheme for a ray that travels over a number of pixels (vertically aligned) that contain sorted intervals.

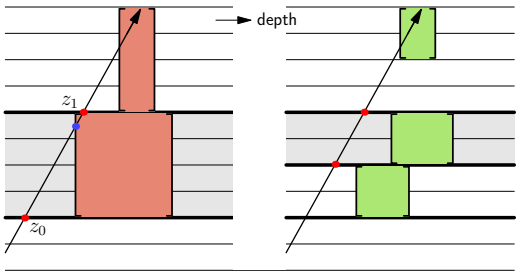

Figure 6: When multi-level-traversal visits a pixel, all intervals therein are checked for overlap with the ray using the perspectively-correct interpolated z projection over the pixel's footprint, z_0 and z_1. If an intersection is found on a higher level (here level 2) the screen-space intersection-point (blue) is used to continue traversal (i.e. here we skip the lower level-1 pixel in the right figure as we know there is no overlapping interval with the ray).

The numbers shown at each interval give the traversal-order; gray intervals are not considered as a previous interval was found to be behind the ray, already.

5.2 Multi-Level Traversal

As can be seen from the example illustrated in Figure 5, even for short rays (in this case terminating after 7 pixels rasterized vertically via DDA), many intervals have to be checked due to the deep nature of our structure. One apparent optimization would be to use binary search on the interval-arrays, instead of simple depth-first traversal. However, given that all rays originate from visible geometry, only a small fractions of rays will travel much behind the front-most layers [Widmer et al. 2015]. Our tests confirmed these results: on average 1.6 intervals are checked per visited pixel, see Section 6.

A common problem with ray marching is, that long rays are very inefficient, especially when only stepping though empty regions [Widmer et al. 2015]. Adding a stride parameter [McGuire and Mara 2014] helps, but scales only linearly and might compute highly incorrect results for larger strides. See Section 6 for a comparison of image quality under different stride-settings. With respect to this, our approach is more along the lines of Widmer et al. [2015], in that we also follow a hierarchical approach.

Based on the multi-resolution interval hierarchy described in Section 4.3, we start the traversal of a ray at the base-level (finest resolution). When we have not found an intersection after n_{up} steps, we switch to a higher level of our hierarchy (coarser resolution). With the traversal as described in Section 5.1 this is straightforward: we simply intersect larger screen-space bounds with the ray and traverse the corresponding higher-level arrays. For each n_{up} successive pixels traced without finding an intersection we repeatedly switch to higher levels. Naturally, we only select higher levels when the coarser pixel's size does not overlap the current level's pixel size, to avoid self-intersection. In contrast to traversal using only a single level of our hierarchy (as described in Section 5.1), we do not terminate traversal when the ray is found to intersect an interval. Instead, if we find an intersection at a higher level, we advance the ray to the hit-point and descend one level. Note that, at this step, we can skip over pixels: we advance the traversal-state to the starting-depth of the interval and re-compute the screen-space position. Figure 6 illustrates this for a ray travelling up in screen-space. When entering the level-2 pixel (left, shaded) we find that the ray overlaps an interval (red). Re-projecting the intersection depth (blue dot) yields a level-1 screen-space position. As the intersection is above the lower level-1 pixel (right panel) we know that there can be no intersection for it and skip forward to the upper one (shaded).

This skipping scheme suggests that descending more than one level at an intersection might be beneficial as the traversal approaches faster to the underlying level-0 interval that represents the sampled scene geometry. However, this would also cause more steps on lower levels for grazing intersections that do not contribute a level-0 intersection, such as the one shown in Figure 6. Overall we found that descending one level at a time performs best.

5.3 Efficient GPU Implementation

The traversal state of our multi-level DDA is only tied to the ray's t parameter, screen-space position and current level, L. Ray/pixel intersections are always expressed relative to t and ray/interval intersections to the derived parameters z_0, z_1 (see Figure 6). Therefore, there are no branches on ray direction, in fact, except for the loops over pixels and intervals there are only trivial branches, (e.g. min/max, no else-clauses).

However, imbalances in ray length can cause compute-units to go under-utilized. To this end we keep track of the number of active rays per warp and prematurely terminate rays when only 4 of a warp are still tracing. This introduces an error, but interpolation from neighboring rays and, if necessary, environment lookup are common approximations in such scenarios [McGuire and Mara 2014]. This scheme is conceptually similar to dynamic fetching in GPU ray tracing [Aila et al. 2012], but without fetching new rays. We believe that having a global ray-queue could be beneficial, but have not investigated in this direction. Note that this scheme is supported in OpenGL with the `GL_ARB_shader_ballot` extension.

When tracing multiple bounces (or samples in general) we can fully exploit this scheme and reload rays as soon as only a fraction of the warp is active. Note that this effort has to be balanced with overhead of computing bounces (or new sample directions). We found the best configuration to generate new rays when over 25% of the treads in a warp are idle. The average speed-up of this scheme is 32% for early termination of single-bounce rays and 10% for multiple bounces (as we also shade at ray termination this impacts the benefit of this optimization).

For multiple bounces our experiments showed that terminating a group of rays as soon as only 4 active rays remain is overly aggressive. It works well for single bounces, but with multiple bounces it causes temporal noise. We found that terminating singular rays still improves performance, but without introducing noticeable noise, as can be seen in Figure 7.

6 EVALUATION

We evaluate our method on a different use-cases where screen-space ray tracing can be applied: reflection, refraction, ambient occlusion (AO) and shadows. While reflection and refraction have similar characteristics (long rays for a part of the visible surfaces)

Figure 7: By terminating long rays, a performance versus quality trade is achieved. The above images show the ground truth with no terminated rays (6ms, left), the difference, multiplied by 8, to terminating up to 1 ray per warp (5.4ms, middle) and up to 4 rays per warp (4.7ms, right).

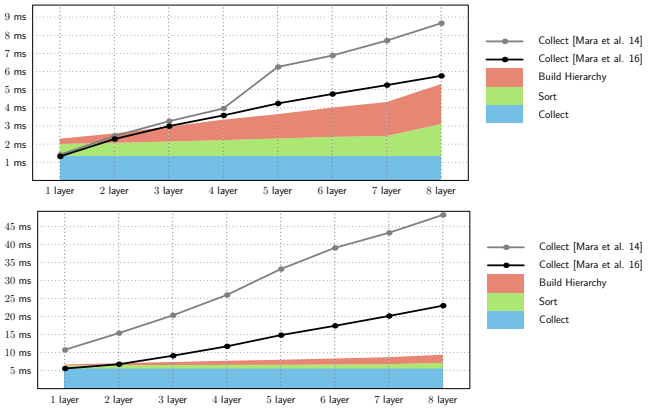

Figure 8: Construction times for multi-layer SSR (lines) and the stages of our multi-layer hierarchy for single-bounce reflections in Sponza (top) and San Miguel (bottom), see also Figure 1, graphed for different numbers of layers collected during rendering. Rendered on a Geforce 1070 at 720p.

ambient occlusion and shadow computations are rather different. For AO the rays cast are usually rather short, and for shadows from point light sources rays have to be cast from almost all pixels of the current view.

We compare the image quality provided by our method against a recent, fast screen-space ray tracing method that follows a similar approach to ours [McGuire and Mara 2014]. Regarding rendering performance we compare our method to the aforementioned screen-space ray tracer [2014], and also include improvements presented in later work [Mara et al. 2016]. To be more concise we refer to this method as SSR, to our method as HSSR (hierarchical screen-space ray tracing). Note that we also provide further data about construction and trace times in 720p and 1080p resolution, as well as GLSL source code in our supplemental material.

Data Structure Generation Performance. In contrast to SSR, we not only have to render multiple layers of the scene, but we also have to build the hierarchy on top of this information. This suggests that HSSR-generation is a more expensive step. However, as indicated by Figure 8, our method catches up with the depth-peeling based SSR even when only few layers (3 for Sponza, 2 for San Miguel) are collected. Note that we have to collect all layers

Resolution	Base Lists	Arrays	Hierarchy	Total	SSR
720p	96	58	40	194	42
1080p	217	132	89	438	95

Figure 9: Overall memory consumption in MiB: Keeping lists and converting to arrays imposes a strong memory overhead. Data for 4 layers and 5 levels of our hierarchy on Sponza.

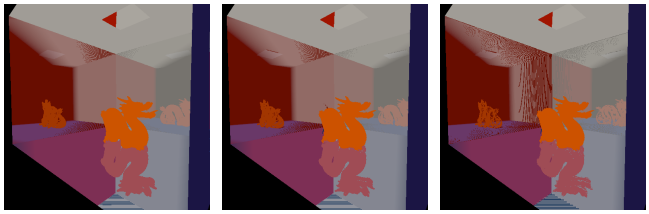

Figure 10: Missed hitpoints: Our HSSR shows only few misses by virtue of DDA (left). A stride of 1 with SSR also yields good results, but stride 4 shows very strong artifacts.

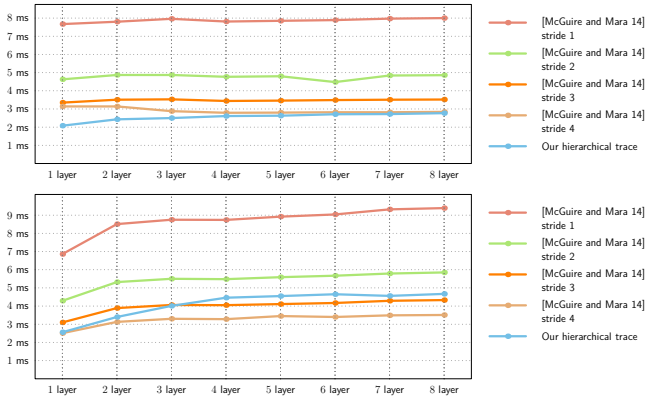

Figure 11: Ray traversal times for SSR under different stride values as compared to our HSSR for single-bounce reflections in Sponza (top) and San Miguel (bottom), see also Figure 1. Rendered on a Geforce 1070 at 720p.

in any case (indicated by the constant overhead shown in blue) and only sorting and hierarchy construction scale with the number of layers. Furthermore, when comparing against SSR for simple collecting-performance, the hierarchy construction (red) should not be taken into account. The times given in Figure 8 show that collecting all fragments in per-pixel lists is, on current hardware, faster than (highly optimized) depth-peeling, even when only few layers are to be computed. This finding might carry over to improving SSR, even though first-layer traversal performance could be impacted (due to less efficient caching by not storing the front-most layer in a single texture). Note that we provide construction times for 1080p in our supplemental material.

Regarding memory requirements our method is more taxing than SSR, as illustrated by Figure 9. The initial list data holding all

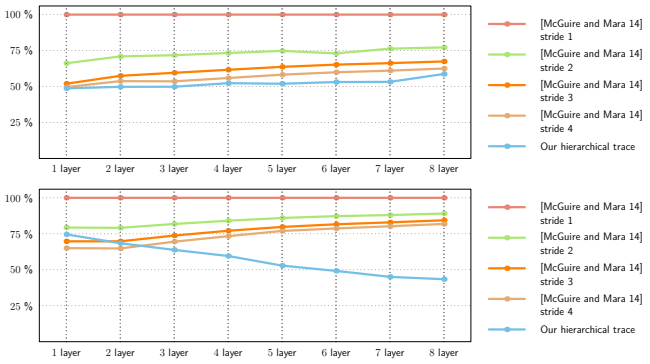

Figure 12: Hierarchy construction and single-bounce reflection times for SSR under different stride values as compared to our HSSR in Sponza (top) and San Miguel (bottom), normalized to SSR with stride 1. See also Figure 1, rendered on a Geforce 1070 at 720p.

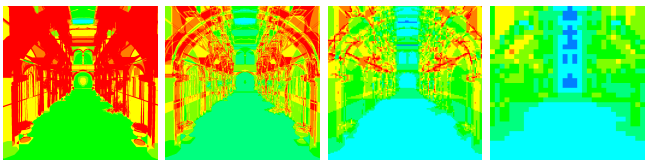

Figure 13: Population of hierarchy levels 0 (left), 1 (middle left), 3 (middle right) and 5 (right) for $\tau = 25$ in Sponza, mapped from 1 (blue) to 8 (red). Depth complexity is continuously reduced for every hierarchy level, therefore runtime memory requirements can safely be assumed to be lower than the worst-case.

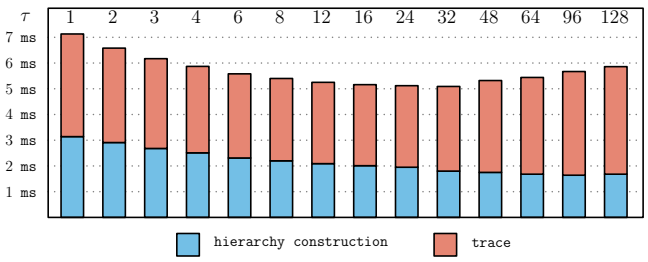

Figure 14: The effect of τ on performance regarding hierarchy construction (red) and trace (blue), ranging from $\tau = 1$ to $\tau = 128$ on a logarithmic x-axis. We found a robust sweet spot ranging from 15 to 35 for all employed scenes.

fragments in the frustum produces an overhead. We note that this overhead scales with the scene's depth complexity and is thus, in contrast to SSR, not decoupled from scene geometry. This could be countered by also using depth-peeling to generate our hierarchy without affecting traversal performance (see below). Therefore our technique to collect the scene fragments should be seen as a different approach that applies to both, SSR and HSSR.

As it is not guaranteed that each new level is of reduced depth complexity, the worst-case memory requirements for 5 levels of our

Figure 15: Reflections in San Miguel, traced at 720p. Trace time for two bounces: 8.3 ms, one bounce down 4.6 ms.

hierarchy are 147MB at 720p and 332MB at for 1080p resolution, assuming 4 full base layers and no merged intervals. Figure 9 shows the amount of memory allocated for holding the hierarchy, which was used in our experiments and found sufficient for all scenes employed, following the same allocation principles as for arrays in Section 4.2. However, when using $\tau \approx 25$, a reduction of depth complexity is very likely and therefore in practice less memory needs to be allocated. For Sponza, this is shown in Figure 13. Also, the effect of the parameter τ on hierarchy construction and trace performance is shown in Figure 14. .

Image Quality. The quality of images computed by screen-space ray marching, when compared to approaches based on the actual scene geometry, is dubious at best. However, appealing results (even when deviating from the correct solution) can be achieved and the SSR-method is well established in practice. With HSSR we provide an alternative to overly aggressive stride settings by skipping over empty space in a less error-prone way. In fact, relying on DDA-style traversal [Amanatides and Woo 1987] our images show fewer artifacts for corner cases. This is illustrated in Figure 10, along with the impact of stride 4 in SSR (a setting that often results in similar traversal times to HSSR).

Ray Traversal Performance. Ray traversal performance using our method is in most cases (see below) much faster than with previous approaches. Figure 11 compares our method's performance with SSR at different stride values. The impact of striding can clearly be seen, however, in contrast to using HSSR it comes at a cost in image quality. In contrast, our method performs at the speed of stride 3 or 4 while not compromising on quality. The reason traversal performance does not suffer from having more layers is, that on average only the front-most fragments are visited. Note, however, that not having multiple layers leads to strong artifacts [McGuire and Mara 2014]. These artifacts, along with artifacts from linked list truncation, can be seen in Figure 16. Still, over all traced samples for Figure 1 (right) both SSR and HSSR visit 1.6 fragments on average, per-pixel. The difference in render times is also visible in the number of iterations required by the DDA. For Figure 1 (right), SSR requires 142.5 iterations on average, whereas HSSR only takes 21.8.

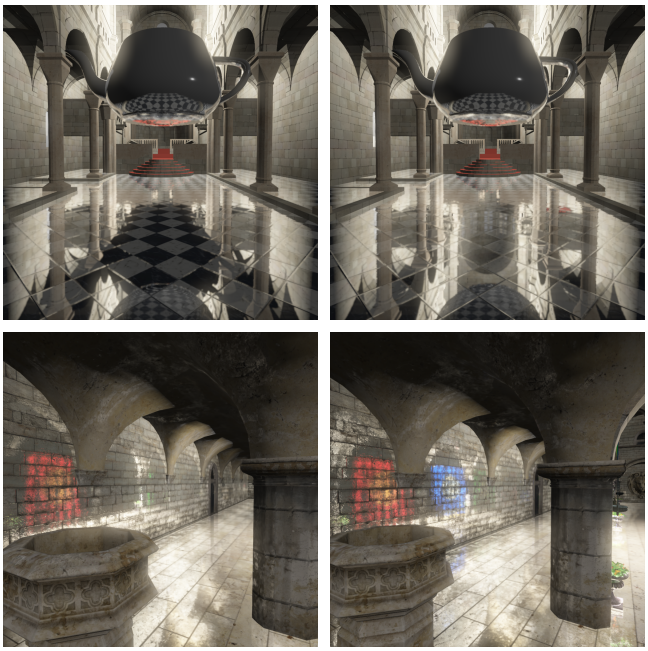

Figure 16: Artifacts from missing scene information when tracing specular reflection rays on a single depth layer (top left) vs. multiple depth layers (top right). In corner cases, truncation of long per-pixel linked lists may lead to geometry misses. Note the missing reflection of the blue curtain in the bottom left image. In the bottom right image, due to a slightly shifted view angle and thus resulting in shorter per-pixel lists, the blue curtain is still found.

The overall time of hierarchy construction and tracing of a single bounce is given in Figure 12, normalized to SSR with stride 1. The figure shows that, for larger numbers of layers, the slower depth-peeling step of SSR pushes the relative performance of HSSR with larger scenes (such as San Miguel in the bottom panel). This effect diminishes when more samples are traced, making the overall performance approach the ray traversal performance (see Figure 11).

Figure 17 shows that our method also works well when including refractions. The image shows reflection rays from the floor that refract (twice) though the bunny, as well as direct refraction rays that make visible the carpeted stairs in the back.

Note that our comparison to SSR uses the shader code published by McGuire and Mara [2014] to ensure a fair evaluation. See our supplemental material for both, SSR and HSSR shader code, as well as for ray traversal performance at 1080p.

Computing AO. The previous measurements have all been on reflection and refraction rays. For AO our hierarchy is not as efficient as the rays are generally very short, which means that it cannot unfold its full potential. For the scene shown in Figure 18 tracing 16 samples with HSSR takes 18.4 ms, whereas SSR only takes 12.1 ms. This can be countered to some degree by not generating as many sub-resolution representations in the interval-hierarchy, but the overhead of switching layers, and of the more exact DDA limit the effect of this. When taking care of all these limiting factors, HSSR has been reduced to SSR, which simply performs better for short rays. Note that for AO there may generally be better screen-space based approximations available [Mara et al. 2016] than using ray traversal.

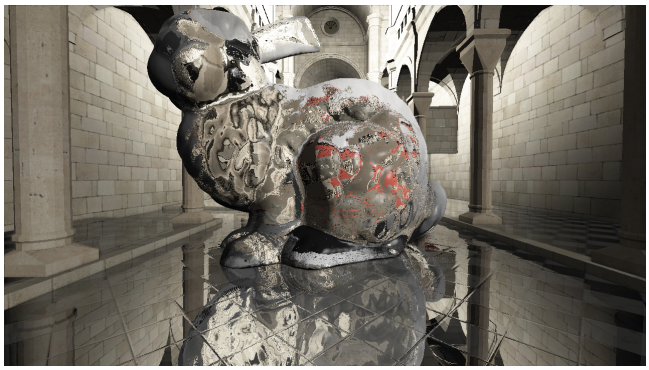

Figure 17: Refraction and reflection: The floor reflects rays that refract (twice) through the bunny (trace time 11.5 ms).

Computing Shadows. Computing shadows from multiple lights (or sampling area lights) is also possible using multi-layered screen-space ray tracing. However, with shadows the limitations of this approach become more apparent: as all visible surfaces have to be checked for shadows, z-thickness becomes more of an issue as rays can become completely orthogonal to the viewing direction. But still, for light sources inside the view-frustum good results can be obtained, as exemplified by Figure 19. Trace times to compute shadows from three point lights for this setup are 24.4 ms with SSR and 10.1 ms with HSSR, at 720p. Nevertheless, shadows are not very stable, tear-up easily and these situations are not as easily covered up as with, e.g., reflections and AO.

Further Optimizations. The measurements above have all been with respect to full resolution data structures and traversal. Common practical optimizations to SSR are sub-resolution data structures and sub-sampling in screen-space, as well as larger stride values. The first two of these can easily be applied to our method, and our supplemental material shows how performance improves when tracing at a lower resolution (while still having full-resolution data). It also contains construction times at 1080p, so that the impact of having a sub-resolution data structure becomes apparent.

Adjusting a stride-parameter is, however, not as simple with our method as striding is replaced by the hierarchy. To this end we have included detailed trace times with varying stride-values above.

7 CONCLUSION

In this paper we have presented a hierarchical extension to multi-layer screen-space ray tracing. We first evaluated a method to construct multi-layered framebuffers alternative to depth-peeling based approaches. We further described our hierarchical multi-layer ray marching scheme and demonstrated that it performs at the level where recent, state-of-the-art methods have to sacrifice image quality to get to. We have also shown the cases in which these claims are to be understood: for long screen-space rays. As demonstrated, our method does not work well, e.g., for ambient

Figure 18: Ambient occlusion computed by tracing 16 samples via HSSR at 18.4 ms, via SSR 12.1 ms.

Figure 19: Shadows from 3 point lights traced by screen-space ray marching at 10.1 ms (HSSR) and 24.4 ms (SSR).

occlusion where only short rays are traced. In general we have presented construction and traversal times without common shortcuts used, e.g., in games, but have noted where they are applicable. We believe this presentation to give a more consistent overview of the performance that can be expected.

In further work it would be interesting to investigate how more orthogonal rays can be kept from traversing in-between fragments that actually represent closed geometry. It would also be interesting to find a balanced compromise between depth-peeling and per-pixel lists, to reap the benefits of both, maybe along the lines of k-buffers.

REFERENCES

Timo Aila, Samuli Laine, and Tero Karras. 2012. *Understanding the Efficiency of Ray Traversal on GPUs – Kepler and Fermi Addendum.* Technical Report NVR-2012-02. NVIDIA Corporation.

John Amanatides and Andrew Woo. 1987. A Fast Voxel Traversal Algorithm for Ray Tracing. In *In Eurographics '87*. 3–10.

Louis Bavoil and Kevin Myers. 2008. *Order Independent Transparency with Dual Depth Peeling.* Technical Report. NVIDIA Corporation.

Cass Everitt. 2001. *Interactive Order-Independent Transparency.* Technical Report. NVIDIA Corporation.

Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata. 1986. IEEE Computer Graphics and Applications. Computer Science Press, Inc., New York, NY, USA, Chapter ARTS: Accelerated Ray-tracing System, 16–26.

Per Ganestam and Michael Doggett. 2015. Real-time Multiply Recursive Reflections and Refractions Using Hybrid Rendering. *Vis. Comput.* 31, 10 (Oct. 2015), 1395–1403.

Graham. 2010. Screen space reflections. (2010). https://forum.beyond3d.com/threads/screen-space-reflections.47780/ Online; accessed June-2017.

Tero Karras. 2012. Maximizing Parallelism in the Construction of BVHs, Octrees, and K-d Trees. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics (EGGH-HPG'12).* Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 33–37.

Christian Lauterbach, Michael Garland, Shubho Sengupta, David Luebke, and Dinesh Manocha. 2009. Fast BVH Construction on GPUs. *Computer Graphics Forum* 28, 2 (2009), 375–384.

Sungkil Lee, Elmar Eisemann, and Hans-Peter Seidel. 2009. Depth-of-field Rendering with Multiview Synthesis. *ACM Trans. Graph. (Proc. of SIGGRAPH Asia)* 28, 5 (2009).

Sungkil Lee, Elmar Eisemann, and Hans-Peter Seidel. 2010. Real-time Lens Blur Effects and Focus Control. *ACM Trans. Graph.* 29, 4, Article 65 (July 2010), 7 pages.

Sungkil Lee, Gerard Jounghyun Kim, and Seungmoon Choi. 2008. Real-Time Depth-of-Field Rendering Using Point Splatting on Per-Pixel Layers. *Computer Graphics Forum* (2008).

Michael Mara, Morgan McGuire, Derek Nowrouzezahrai, and David Luebke. 2014. *Fast Global Illumination Approximations on Deep G-Buffers.* Technical Report NVR-2014-001. NVIDIA Corporation. 16 pages.

Michael Mara, Morgan McGuire, Derek Nowrouzezahrai, and David Luebke. 2016. Deep G-Buffers for Stable Global Illumination Approximation. In *HPG*. 11.

Morgan McGuire and Michael Mara. 2014. Efficient GPU Screen-Space Ray Tracing. *Journal of Computer Graphics Techniques (JCGT)* 3, 4 (9 December 2014), 73–85.

Quirin Meyer, Jochen Süßmuth, Gerd Sußner, Marc Stamminger, and Günther Greiner. 2010. On Floating-Point Normal Vectors. *Computer Graphics Forum* 29, 4 (2010), 1405–1409.

Oliver Nalbach, Tobias Ritschel, and Hans-Peter Seidel. 2014. Deep Screen Space. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '14).* ACM, New York, NY, USA, 79–86.

Greg Nichols and Chris Wyman. 2009. Multiresolution Splatting for Indirect Illumination. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games (I3D '09).* ACM, New York, NY, USA, 83–90.

Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. 2000. Surfels: Surface Elements As Rendering Primitives. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '00).* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 335–342.

Tobias Ritschel, Thorsten Grosch, and Hans-Peter Seidel. 2009. Approximating Dynamic Global Illumination in Screen Space. In *Proceedings ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games.*

Kai Selgrad, Carsten Dachsbacher, Quirin Meyer, and Marc Stamminger. 2014. Filtering Multi-Layer Shadow Maps for Accurate Soft Shadows. *Computer Graphics Forum* 34, 1 (2014), 205–215.

Kai Selgrad, Christian Reintges, Dominik Penk, Pascal Wagner, and Marc Stamminger. 2015. Real-time Depth of Field Using Multi-layer Filtering. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games (i3D '15).* ACM, New York, NY, USA, 121–127.

Jonathan Shade, Steven Gortler, Li-wei He, and Richard Szeliski. 1998. Layered Depth Images. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '98).* ACM, New York, NY, USA, 231–242.

Peter Sikachev. 2014. Reflection System in Thief. In *SIGGRAPH Courses (2014).* ACM.

Tiago Sousa, Nick Kasyan, and Nicolas Schulz. 2011. Secrets of CryENGINE 3 Graphics Technology. In *SIGGRAPH Courses (2011).* ACM.

Tomasz Stachowiak and Yasin Uludag. 2015. Stochastic Screen-Space Reflections. In *SIGGRAPH Courses (2015).* ACM.

Michal Valient. 2014. Reflections and Volumetrics of Killzone Shadowfall. In *SIGGRAPH Courses (2014).* ACM.

Sven Widmer, Dawid Pajak, A. Schulz, Kari Pulli, Jan Kautz, Michael Goesele, and David Luebke. 2015. An Adaptive Acceleration Structure for Screen-space Ray Tracing. In *Proceedings of the 7th Conference on High-Performance Graphics (HPG '15).* ACM, New York, NY, USA, 67–76.

Bartlomiej Wronski. 2014. Assassin's Creed 4: Black Flag Road to next-gen graphics. (2014). GDC Talk.

Chris Wyman. 2005a. An Approximate Image-space Approach for Interactive Refraction. *ACM Trans. Graph.* 24, 3 (July 2005), 1050–1053.

Chris Wyman. 2005b. Interactive Image-space Refraction of Nearby Geometry. In *Proceedings of the 3rd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia (GRAPHITE '05).* ACM, New York, NY, USA, 205–211.

Jason C Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. 2010. Real-Time Concurrent Linked List Construction on the GPU. *Computer Graphics Forum (Proc. EG Symposium on Rendering)* 29, 4 (2010), 1297–1304.